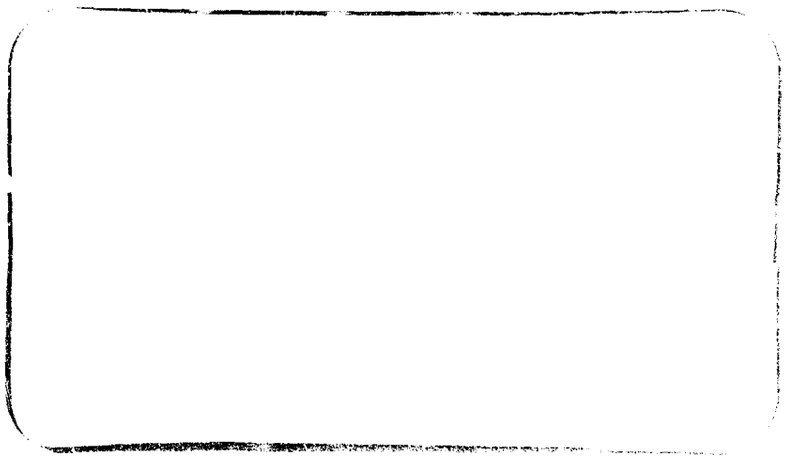

Department of Psychology
University of Washington • Seattle

17

THE FORTRAN DEDUCTIVE SYSTEM

J. R. Quinlan and E. B. Hunt

This research was partially sponsored by the National Science Foundation, Grant GB-1438-R and partially by the Air Force Office of Scientific Research, Office of Aerospace Research, United States Air Force, under AFOSR Grant AF-AFOSR-1311-67.

Department of Psychology
University of Washington--Seattle

January 24, 1968

The Fortran Deductive System

J. R. Quinlan and E. B. Hunt
The University of Washington

The Fortran Deductive System (FSD) is a program which solves generalized theorem proving problems. It receives as input the definition of a deductive system (algebra, geometry, first order predicate calculus, etc.), then a series of theorems. The program attempts to prove each theorem, using as premises the axioms of the system and any previously proven theorems which the program user has ordered it to retain. The notation used by the program to define a particular deductive system is very similar to that used in conventional school mathematics. This makes it possible for a person to use the system even though he has no knowledge of FDS' internal structure. In fact, most users of the system do not have this knowledge.

The FDS was developed as a vehicle for studying two problems in computer science, both of which are closely related to analagous problems in the behavioral sciences. The major use of FDS has been to study "pure cases" of problem solving processes. By observing the performance of well defined algorithms for solving symbolic problems, we hope to develop an empirical typology of problem types such that, knowing the type of a particular problem, we will be able to predict the appropriate method for solving the problem before we do, in fact solve it. A somewhat simpler statement of this goal is that we want to know what certain problem solving algorithms work well. A second research goal is the augmentation of human problem solving

ability. It is obvious that if we could produce a theorem prover more effective than man, the range of problems which could be attacked would be increased. In addition, by exploring the potentials of particular computer-executed problem solving algorithms we may assist human problem solvers in two other ways. We might discover problem solving methods which, in addition to being machine executable, could be taught to humans with a resultant increase in their intellectual skills. By extending the range of problems which computers can handle, we can increase the potential contribution, and alter the work distribution, in man-machine problem solving teams.

The FDS is an intellectual descendant of the General Problem Solver (GPS) of Newell, Shaw and Simon (1959; Newell and Simon, 1961), from whom we obtained the original idea and an orientation for our early efforts. By now, however, the relationship has become quite distant, as FDS incorporates several concepts and programming techniques which are not contained in GPS, and similarly, some of the GPS capabilities have been dropped. FDS also incorporates many ideas taken from recent work on the construction of generalized compilers ("compiler-compilers"). As its name implies, the program is written entirely in FORTRAN IV. ALGOL versions have also been implemented. We have not found any substantial restrictions due to limitations of these languages.

A formal definition of FDS is available (Quinlan and Hunt, 1967). It, and program listings if desired, can be obtained from us. The program has been running successfully

for over a year. To our knowledge, runs have been executed on the IBM 7094, the CDC 3600, the Burroughs B 5500, and the English Electric KDF-9. In this paper we shall give an informal description of the general features of the program, and discuss a few selected applications to give some idea of its range and power.

From the user's point of view

The user can regard FDS as an "alterable" black box which accepts theorems as input and produces solutions as output. It is alterable in the sense that prior to the first theorem the user specifies what a well formed expression looks like, what notation he will use, and what rules of inferences are permissible.

Notation: The user first indicates the names of his operands and operators. He can specify non-numeric constants (TRUE, FALSE, GOOD, BAD), variables (X, Y, WHO, WHICH), unary connectives (NEG, NOT, LOG, BELOW), and binary connectives (+, -, NOT, OR, BELIEVES). In addition, the program recognizes the integers.

Axioms: Excepting the integers, the program contains no inherent definition for the user's notation. Meaning is established when the user defines his axioms. Each axiom is stated as a rewriting rule, i.e., structure A may be rewritten as structure B. We will use ":=" to mean "may be rewritten as," although in fact this symbol may be specified by the user. Axioms are presented to the system in the conventional, or infix, notation. For example, the commutivity property of "+" in algebra would be expressed

as

$$A + B := B + A.$$

This rule informs the system that any two well formed formulae separated by a "+" may be rewritten with the order of the formulae reversed. As in conventional algebra, a well formed formula is any variable or constant, any unary operated followed by a well formed formula, or any two well formed formulae separated by a binary operator. Parentheses must be used to supply information concerning the order of priority of operators where this is ambiguous, in the same manner as they are used in school algebra.

System parameters. As an option, the user may supply the program with some "hints" about the difficulty of a problem, in terms of the expected number of steps to a proof, how large the expressions should grow to in intermediate stages of proof, and so forth. These parameters may affect the power of the system in some situations, in others its performance is surprisingly independent of parameter settings.

Problems: Problems are input in exactly the same form as axioms, i.e., a well formed formula on the left to be rewritten as on the right of a rewriting symbol. Optionally, the user may indicate that if a particular theorem is proven, it is to be used in the proof of subsequent theorems.

Solutions

A theorem proving problem is solved by FDS when the program finds an ordered set of rewritings which change one string of symbols (the "left-hand side" above) into another (the "right-hand side"). As an example, suppose we have the axioms

1. $A + B := B + A$

2. $(A + B) + C := A + (B + C)$

and the problem

$$A + (B + C) := (A + B) + C.$$

This is a problem because the rewriting symbol is not reflexive ...i.e., it acts more like the "implies" of logic than the "=" of algebra. The following solution would be obtained by the FDS:

INITIALLY		$A + (B + C)$
USING AXIOM	1	$(B + C) + A$
USING AXIOM	1	$(C + B) + A$
USING AXIOM	2	$C + (B + A)$
USING AXIOM	1	$(B + A) + C$
USING AXIOM	1	$(A + B) + C$ SOLUTION

This simple example illustrates two points. Nowhere has the program gone beyond the semantic meaning of the symbols implied in the axioms. The "cleverness" of the program is solely in deciding which axioms to apply, and whether to apply them to the entire string of symbols which constitute the present state-- each successive line of the proof--or to some substring of the present state.

Program Organization

The FDS internal operation will now be described briefly. Internally, formulae are expressed in the suffix, or "Polish" notation. A well formed formula is defined as (a) a constant or variable, (b) a unary operator followed by a well formed formula, or (c) a binary operator followed by an ordered pair of formulae. In the last case, we have

followed the convention that the operand which would be the right-hand operand in the normal notation is expressed first. As an example, if the user states A + (B + C), FDS will store the string ++CBA. It is important to note that any operator symbol will head a substring which is itself a well formed formula. Also, the order of the symbols in the FDS string will unambiguously specify a tree structure for an expression, i.e., a graph in which each node represents an operator or a variable or constant, and those nodes which represent operators have below them subgraphs specifying their operands. The GPS program achieves an identical representation using list processing techniques.

A problem is a command to FDS to rewrite a given suffix string, called the state, and corresponding to the left-hand side of the problem in the external representation, into another suffix string, called the goal, which corresponds to the right-hand side of the external representation. The two strings are compared symbol by symbol. If they are identical, then the problem is solved. It is retranslated to the external representation and printed. If the two strings are not identical, a difference set is established between them. This is done by starting at the leftmost symbol (the main operator of the expression), and comparing corresponding symbols. When a difference is noted, the pair of symbols which gave rise to the difference are stored on the difference set, and comparison continues, but does not include any symbols which express operands of the symbols which gave rise to the difference pair.

An example may help clarify this. Returning to the simple problem given above, we see that the internal representation of axioms 1 and 2 in FDS would be

1. +BA := +AB

2. +C+BA := ++CBA

while the problem is

++CBA := +C+BA.

The first difference pair is the pair (+,C), where the + is the second + in the state (left-hand) string, and the C is the C in the right-hand string. Since the + begins a substructure (corresponding to (B + C) in the external problem), the substring +CB on the left is skipped. Thus, the second difference pair is (A, +), where the + is the second + of the goal string. This in turn, begins the substructure +BA (A + B), so the remainder of the goal string is skipped. Therefore, these two difference pairs comprise the difference set.

The difference set states, in effect, what must be changed into what in order for the goal to be achieved. The FDS then examines both the axioms and the state and goal strings in order to determine a set of pairs (c,d), where c is an axiom and d specifies a substring of the state string. That is, c is a rewriting rule and d is the place in the state string where it will be applied. Let us call this an operator. In the example, the first step applied axiom 1 to the entire string, while the second step applied the same axiom to a subexpression.

In applying an operator it may be found that the state

string (or substring) is not in a form appropriate for the axiom's application. In this case a check is made to determine how many changes in the state string will be required before the axiom can be applied. If the system decides to apply an axiom which requires a change of the state string, the subproblem of making this change in the state string will be attacked. Those familiar with the GPS literature will recognize this as being very similar to the "means-end analysis" and recursive problem solving technique used in the Newell, et. al. program. The major difference between FDS and GPS at this point is that in FDS subproblems are not attacked by recursive application of the program, but instead are attacked in a way that insures that when there are several alternative solutions to a subproblem, an appropriate solution for solving both the subproblem and advancing the solution of the main problem will be found.

The performance of the program depends heavily upon its ability to determine which operator to try first. This information is obtained by two subprograms, both of which are replaceable modules of the main program. One subprogram "looks ahead" in the sense that it determines how many useful changes will be achieved by a given operator. The assumption is made that any subproblems which may arise can be solved. A second subprogram compares the operations recommended by the first subprogram to a record of previously successful applications of axioms. Each operator is classified into one of twenty categories, based on the structure of both the axiom and the state to which it is to be applied.

The operator is then assigned a rating determined by the ratio of the number of times operators in its class have lead to solutions to the total number of times they have been tried. These frequencies are built up as the program obtains experience in solving problems in a particular deductive system. Simple minded as the learning algorithm is, our experience has been that it is quite effective in improving the program's performance.

Applications of the FDS

We have conducted a reasonably large number of studies of the program's performance in a variety of areas of mathematics. To give some idea of the power of the system, the results of these studies will be summarized briefly.

Algebra. The system is a quite powerful manipulator of conventional algebraic formulae. Over a hundred theorems have been proven concerning the manipulation of variables under the operations of addition, subtraction, negation, and multiplication. The average time per theorem is about twenty seconds.

We do not present FDS as a simulation of human thought. In fact, in algebra it is a somewhat better problem solver than most people. To test this assertion, we used FDS to solve twelve problems based upon the following restricted axioms of algebra:

1. $A + B := B + A$
2. $(A + B) - B := A$
3. $A := (A + B) - B$
4. $(A - B) + C := (A + C) - B$

5. $A + (B + C) := (A + B) + C$

6. $(A + B) - C := (A - C) + B$.

The same problems were presented to thirty undergraduate psychology students at the University of Washington, and in thirty minutes they produced an average of two solutions each. The program solved all twelve problems in less than five minutes. One problem, to prove

$$(A - C) - (B - C) := A - B,$$

was not solved by any of the students. Informally the same problem was attempted by several professors..including mathematicians and mathematical psychologists..and graduate students in mathematics, engineering, and computer science. Only one student, a senior in mathematics, produced any solution, and his was different from that produced by FDS. By contrast, some problems on which FDS expended a fair amount of time (by its standards) proved quite easy for people.

Logic: FDS was used to solve problems selected from the exercise sets in an elementary school "new mathematics" text book, Suppes and Hill's Introductory Logic for Schools (19). Although some problems were found for which an FDS formulation was clumsy, no problems were found in this book which the program could not solve.

Trigonometry: A number of informal studies were carried out in which the program was used to solve identity proving problems in trigonometry. These studies illustrated an important principle, the program is flexible enough so that how the user represents the deductive system will exert

a considerable influence on the difficulty of obtaining a solution. Several alternate representations are possible for trigonometry. Depending on which representation was used, a problem might be either hard or easy. It has been noted that this occurs in other problem solving programs (Ernst and Newell, 1967).

Pattern identification: It has been suggested that complex patterns, such as pictures of houses, can be recognized as examples of a generic class by using a simplification system very much like parsing in a phrase structure grammar (Minsky, 1963; Ledley, 1962, 1965). For example, in such a system, a "House" might be defined as a roof on top of wall, where wall could be defined as wall or wall with window or wall with door or wall beside wall, and roof, in turn, be defined as roof or roof with gable or roof with chimney. Even with such a simple "grammar," quite complex pictures can be drawn. When the grammar for recognizing picture classes is input to FDS as the deductive system, the program is capable of recognizing very intricate patterns as examples of a class. The patterns, of course, must first be coded into a symbolic representation of a picture, as the FDS does not at present have any capacity for graphic input of data. In this application the learning procedures described proved noticeably and uniformly effective.

Sanderson algebra: Sanderson (1967) has developed an algebraic representation for flow diagrams which might represent simple computer programs. The motivation for his work was an attempt to develop a deductive system which could be used to prove that two programs, as defined by their flow

charts, were in fact equivalent computations. He proved ten theorems concerning program equivalence using this algebra. The proofs of all ten theorems were reproduced by FDS.

Inequalities: Some attempts have been made to prove the theorems of the calculus of inequalities, using previous results from logic and algebra. In these studies the FDS has had to work with better than 130 previously proven theorems. While some elementary theorems have been proven, the need to search a very large number of rewriting rules to determine the next operation has caused the program to spend a great deal of time on what seem, to humans, to be reasonably easy problems.

Current Status and Future Plans

Thus far, FDS has been applied only to re-prove well known, and to a mathematician elementary, theorems. We hope to extend this somewhat in the next few years by applying FDS to more advanced areas of mathematics. In this work we do not expect the program to exceed the capacity of the intelligent human mathematician. It may, however, serve as a useful complement to him. An interactive version of FDS has been implemented on the Burroughs B-5500 remote access system, and can be used in studies of man-computer problem solving teams.

Considerably more work needs to be done on learning algorithms for the theorem prover. Intuitively, it would seem that experience with theorem proving problems should lead to the knowledge that certain rewriting rules are appropriate in given situations. The question is, can we

write a program which develops these rules of thumb for itself? This is closely related to our interest in problem typology--can we find classes of problems such that certain problem solving procedures are appropriate for all members of the class?

In summary, we have developed a useful, reasonably powerful theorem prover using standard algebraic computing languages. The system is capable of proving theorems which are quite difficult for the average university undergraduate, but it has not yet produced any proofs which would be considered exceptionally good by a professional mathematician. This, of course, is a difficult goal to reach, but we feel that we have made progress towards it.

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author)
University of Washington
Psychology Department
Seattle, Washington

2a. REPORT SECURITY CLASSIFICATION
Unclassified

2b. GROUP
---

3. REPORT TITLE
THE FORTRAN DEDUCTIVE SYSTEM

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)
Scientific Interim

5. AUTHOR(S) (First name, middle initial, last name)
J. Ross Quinlan and Earl B. Hunt

6. REPORT DATE
January 24, 1968

7a. TOTAL NO. OF PAGES
14

7b. NO. OF REFS
8

8a. CONTRACT OR GRANT NO.
AF-AFOSR-1311-67

b. PROJECT NO. ---- 9778-01

c. 6144501F

d. 681313

9a. ORIGINATOR'S REPORT NUMBER(S)
[redacted]

9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)
AFOSR 68-0818

10. DISTRIBUTION STATEMENT
1. This document has been approved for public release and sale; its distribution is unlimited.

11. SUPPLEMENTARY NOTES
------TECH, OTHER

12. SPONSORING MILITARY ACTIVITY
Air Force Office of Scientific Research
Office of Aerospace Research
United States Air Force (SRLB)

13. ABSTRACT

A Fortran Program to solve generalized algebra type problems has been developed. It is a heuristic program, designed to give rapid solutions for problems which it can solve. The program accepts as input a definition of an algebraic system, then solves problems in the system. The program's operation is described here in general terms and examples of its operation are given.

DD FORM 1 NOV 65 1473

Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Algebra						
Problem solving						
Heuristic						
Computer programming						
Artificial intelligence						